VAX C Programming Concepts

Chapter 4

Program Structure

This chapter introduces the basic features of C to the programmer experienced in other languages. The text provides detailed examples and short tutorials, as well as numerous pointers to other chapters in this manual.

A VAX C program is a group of user-defined functions that cannot be nested (you cannot define functions within other function definitions). This chapter also describes VAX C function definitions, function declarations, and the following components of C program structure:

- Function definitions
- Function declarations
- Function prototypes
- Function parameters and arguments
- Program identifiers
- Blocks
- Comments
- VAX C language keywords
- Functionality similar to that provided by lint

4.1 C Programming Language Background

The C language is a general-purpose programming language that is manageable due its small size, flexible due to its ample supply of operators, and powerful in its utilization of modern control flow and data structures. The C language was originally designed and implemented on a UNIX® system using the PDP–11. The designers of the language describe C as follows:

"The [C] language . . . is not tied to any one operating system or machine; and although it has been called a 'system programming language' because it is useful for writing operating systems, it has been used equally well to write major numerical, text-processing, and database programs."[1]

Like assembly language, C was not designed to meet the needs of any particular application. C manipulates and stores data by considering the similarities found in modern machine architecture. However, C is not as complex as assembly language and is not machine dependent. A program is considered portable if

® UNIX is a registered trademark of American Telephone & Telegraph Company.

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice Hall, 1988), p. 1.

you can compile and run its source program using several different compilers on several different machines. C is a language that yields highly portable programs.

There is no ANSI or other industry-wide standard for the C programming language, but there is a consistency of functionality between implementations. This consistency is necessary for C to be portable across systems, which is one of the most desirable features of the language. Not only must C source programs be portable, but the language features themselves must produce the same effects on all systems when you compile and run programs.

C was developed in a UNIX system environment, and was used to rewrite most of that operating system, so many standard methods of operation in C are related to UNIX. For instance, UNIX systems access files by a numeric file descriptor, so C implementations should provide functions to access files by file descriptor. In a UNIX system environment, you can expect a concise command structure, an ability to redirect output from one program or command to the input of another program or command, an ability to create asynchronous and synchronous subprocesses, and an ability to manipulate the operating system features without many restrictions and system safeguards.

Some standard C constructs include preprocessor directives and macros. In a UNIX system environment, a preprocessor completes the tasks designated in the preprocessor directives located in the source code before the compiler takes any action.

4.2 The VAX C Programming Language

The VAX C programming language incorporates the features that are fundamental to the C language and that exist in most C compilers, based on the language design of Brian W. Kernighan and Dennis M. Ritchie. However, this version of VAX C for ULTRIX also incorporates some C language extensions based on the current draft of extensions proposed by the ANSI committee that is reviewing the C language. The added extensions are those most likely to be adopted. Digital does reserve the right to change the VAX C language, particularly these extensions, if the ANSI committee does not adopt the proposed extensions. Finally, VAX C also includes unique features that work directly and efficiently with the VMS operating system environment.

If you decide to retain both portability and utilize the ULTRIX environment, you can use special constructs of VAX C and options of the **vcc** command (such as the VAX C preprocessor predefined symbols and the command line option **–V standard=portable**. These constructs allow you to execute some segments of code only when running on ULTRIX systems, and to execute other segments of code when running on systems other than ULTRIX systems. See Chapter 9 for more information about the preprocessor directives. See Chapter 2 for information about the **vcc** command line.

4.3 Writing Your First Program

Writing your first program in an unfamiliar language can be frustrating, particularly if the program is complex and you introduce numerous errors. Start with simple programs. Computers are efficient at processing numbers, so the first program presented here adds two numbers and stores the total in a variable. Example 4–1 shows how to code such a program in the VAX C language.

Example 4–1: Simple Addition in VAX C

```
❶ /*  This program adds two numbers and places the sum in   *
   *  the variable total.                                   */

❷ main()                          /* The function name "main" */
      {                           /* Begins function body     */
❸        int total;               /* Variable of type "int"   */
                                  /* Blank lines are allowed  */
❹        total = 2 + 2;           /* Answer placed in "total" */
      }                           /* Ends the function body   */
```

Key to Example 4–1:

❶ The text between the character sequences (/*) and (*/) provide comments. You cannot place comments within comments (that is, they cannot be nested), but you can place comments anywhere that white space can appear. White space is an area within the source code where blank spaces or blank lines may separate code. In the following chapters, allowable white space is defined for VAX C constructs.

❷ VAX C programs are comprised of user-defined external functions that cannot be nested. Here, a function named main is defined. In VAX C, execution of a program begins at either a function named main or at a function using the main_program option; if a user-specified main function does not exist, the first function in the program stream at the time external references are resolved is the default main function. The main_program option is VAX C specific and is not portable. For more information about the syntax and using the main_program option, see Section 4.8.1.

VAX C functions have methods of exchanging information using parameters and arguments. In the function definition of main, there are no parameters, as designated by the empty parentheses. In Example 4–1, the function main cannot receive information by means of parameters.

To specify parameters in a function definition, list the parameter identifiers within the parentheses and separate them with a comma (,). You must declare the parameters before the beginning of the body of the function. If you call a function from within function main (you normally would not call the main function from another part of your program), the function name is followed by a list of arguments delimited by parentheses and separated by commas. The number of arguments must correspond with the number of parameters in the function declaration. In Example 4–1, there are no function calls.

The function performs its task as determined by the statements found in the body, and may or may not return a value to the calling expression. The body of the function main is delimited by braces ({ }). They are like the **DO-END** of PL/I or the **BEGIN-END** of Pascal. Usually, the body contains one or more **return** statements. A **return** statement specifies what, if anything, is returned to the expression that called the function. Depending upon the setup of the function, you can omit the **return** statement, and its return value will remain undefined. If a function does not return a value, you can declare the function to be of data type **void**. For more information about functions, see Section 4.10.

❸ The variable total is declared and defined within the function main. You must declare all variables before referencing them within the program. These declarations end with a semicolon (;). If you declare a variable, you specify its data type. Data types specify the amount of storage required and how to interpret the stored object. For example, variable total is of the data type **int** (integer), the object of which requires 32 bits (4 bytes or 1 longword) of memory. VAX C interprets variables of type **int** as integers having a positive or negative sign, or having the value of 0.

When you define a variable, you specify its storage class, which affects its location, lifetime, and scope. Variables of type **int** declared within a function have a default storage class of **auto** (automatic). Variables of this storage class receive storage space when the function is activated and storage is freed when control of the calling function resumes. Not all storage classes are implemented by default. You can specify all VAX C storage classes and may place the reserved word describing the storage class either before or after the reserved word that describes the data type in the variable declaration.

Data types and storage classes are very important when determining the scope of a variable. For more information about data types, see Chapter 7. For more information about storage classes, see Chapter 8.

The reserved words used to identify data types (such as **int** and **double**), storage classes (such as **auto** and **globalvalue**), statements (such as **if** and **goto**), and operators (such as **sizeof**) are called keywords. Keywords are predefined identifiers that cannot be redeclared. You cannot use these words to identify variables and functions in your programs. Keywords must be expressed in lowercase letters. Section 4.13 lists the VAX C keywords.

VAX C is a case-sensitive language. You can declare variables, such as total, in any mixture of upper- or lowercase letters. If you reference the variable total in your program, the reference also must be lowercase. For example, if you try to reference the variable Total, an error occurs; the compiler does not recognize the variable name due to the initial capital letter.

❹ The sum of 2 + 2 is stored in the variable total. This is done using a valid VAX C statement. You can use any valid expression as a statement by ending it with a semicolon (;). The identifier total is a declared variable. The equal sign (=) and the plus sign (+) are valid VAX C operators. The numbers being added are valid constants. For more information about the various VAX C statements, see Chapter 5. For more information about the VAX C operators, see Chapter 6.

4.4 Producing Input/Output

The C language includes no facilities to administer input and output (I/O). However, all implementations must have methods that allow programs and users to communicate. The lack of communication in Example 4–1 is inconvenient; there is no way to know if the program assigns the correct value of 4 to the variable total. You can use an ULTRIX System Library function to output the value of the variable total to the terminal. For more information about the ULTRIX System Library functions, see the *ULTRIX Documentation Set*.

Before you can execute any of the example programs in this manual, you must define, in the correct order, the libraries that the linker must search to resolve references to library functions. For example, to compile and link a source program that employs functions from the math library, you must link against the ULTRIX System Library by specifying /lib/libm with the **vcc** command line

option **-lm**. For information about linking, see Chapter 2 and the *ULTRIX Documentation Set*.

VAX C macro references within program source code look just like function references. However, the compiler replaces macro references with VAX C source code at an early stage in the execution process. The compiler locates VAX C macro source code in the .h definition files provided with ULTRIX systems. For example, you can display the standard I/O function, stdio.h, at your terminal with the following command:

```
% cat /usr/include/stdio.h RETURN
```

If this command causes an error, see your system manager. It is a good idea to type or print all the .h files to see the macros and definitions provided with ULTRIX systems.

For more information about macros, see Chapter 9.

Example 4–2 shows that by using the ULTRIX **printf** function, a VAX C program can print a message to the terminal.

Example 4–2: Output of Information

```
/*  This program adds two numbers, assigns the value 4 to *
 *  variable total, and then prints the answer on the     *
 *  terminal screen.                                      */

#include <stdio.h>

main()
{
   int total;
   total = 2 + 2;
                              /* Print intro string       */
➊  printf("Here is the answer: ");
   printf("%-d.", total);     /* Print the answer         */
}
```

Key to Example 4–2:

➊ The **printf** function writes to the standard output device (the terminal screen). The first call to the function **printf** passes a string as the argument. The second call to **printf** passes a string with special formatting characters and a variable as arguments. Within the formatting string, the percent sign (%) is replaced by the value of total, the minus sign (–) left-justifies the output, and the letter d forces the value of the argument to be expressed as a decimal number. The period (.) prints immediately after the value of total.

The output from Example 4–2 is as follows:

```
Here is the answer: 4.
```

If you want to print the value of total on a separate line, add the newline character (\n) to the string. Example 4–3 shows how to output on two lines.

Example 4–3: Output Using the Newline Character

```
/*  This program adds two numbers, stores the sum in the        *
 *  variable total, and then prints the answer on two           *
 *  separate lines on the terminal screen.                      */
#include <stdio.h>
main()
{
   int total;
   total = 2 + 2;
                                    /* Print intro string       */
   printf("Here is the answer...\n");
                                    /* Print the answer         */
   printf("%-d.", total);
}
```

The output from Example 4–3 is as follows:

```
Here is the answer...
4.
```

After writing a program that produces output, it is necessary to compile, link, and execute the program using the **vcc** command to see the results. Compiling a program translates the source code to object code. Linking a program organizes storage and resolves external references (for example, references to VAX C functions). Running a program executes the image.

In the ULTRIX environment, a file is distinguished by a file name and by a file extension. Choose a file name that is easy to identify. The file extension should reflect the functionality of the file. For example, the file extension .c is the required source file extension for the VAX C compiler.

After you create and appropriately name your program, invoke the VAX C compiler to compile and link it, then execute the result, as follows:

```
% vcc  -o addition addition.c RETURN
% addition RETURN
Here is the answer...
4.
$
```

You may have to define more libraries to the linker to use C functions in your program. For more information about creating source code and the compilation process, see Chapter 2.

4.5 Controlling Program Flow

There will be occasions when you must execute one or more VAX C statements given a certain condition. There will be other occasions when you must execute one or more VAX C statements repeatedly, within the body of a loop, until you meet a certain condition. There are several statements in VAX C that accomplish these tasks. These statements are the **if** statement, the **switch** statement, the **do** statement, the **while** statement, and the **for** statement. For information about statements that loop until meeting a condition, see Chapter 5.

4.5.1 The if Statement

You can use the **if** statement to force the program to execute one or more VAX C statements when a specified condition exists. Example 4–4 shows a program using the **if** statement.

Example 4–4: Conditional Execution Using the if Statement

```
/*  This program asks the user to guess a letter.  The      *
 *  program tells whether the answer is correct or          *
 *  incorrect.  The program is hard coded to accept 'a' or  *
 *  'A' as the correct letter.                              */

#include <stdio.h>

main()
{
   char ch;                    /* Declare a character      */
                               /* Ask the user to guess    */
   printf("Guess which letter I'm thinking of!\n");

❶  ch = getchar();             /* Get the character        */

                               /* Correct = "a" or "A"     */
❷  if (ch == 'a' || ch == 'A')
                               /* If correct guess         */
      printf("You're right!");
   else                        /* If incorrect guess       */
      {
         printf("You're wrong.\n");
         printf("You'll have to try again!");
      }
}
```

Key to Example 4–4:

❶ The standard I/O **getchar** function retrieves a character from the standard input device (the terminal); the program pauses, waiting for the user to type a character and to press the RETURN key. The **getchar** function retrieves one character and ignores any others that are typed.

❷ If the letter that the user types is either a or A, a message stating that the choice is correct prints. If any other letter is typed, a message stating that the choice is incorrect prints. The equality operator (==) compares the variable ch with the constants `'a'` and `'A'`. The logical OR operator (| |) presents the condition to test. If there is more than one statement to be executed conditionally, you must enclose the statements within braces ({ }). A statement or statements enclosed within braces is called a block or a compound statement. The concept of blocks is important to determine the scope of variables. See Section 4.14 for more information about blocks.

Sample output from Example 4–4 is as follows:

```
% example4 RETURN
Guess which letter I'm thinking of!
B RETURN
You're wrong.
You'll have to try again!
```

4.5.2 The switch Statement

The **if** statement is not the only method of specifying statements to be executed given a certain condition. A **switch** statement can perform the same task as the **if** statement in Example 4–4, but it is particularly useful when many conditions must be tested. Example 4–5 shows a program using the **switch** statement.

Example 4–5: Conditional Execution Using the switch Statement

```
/*  This program plays the same guessing game as the      *
 *  previous example except that it uses a switch         *
 *  statement.                                            */

❶ #include <ctype.h>           /* Include proper module      */

#include <stdio.h>

main()
{
   char  ch;

   printf("Guess what letter I'm thinking of!\n");
   ch = getchar();
   if (isupper(ch));          /* If ch is uppercase         */
❷  ch = _tolower(ch);         /* Convert "ch": lowercase    */
   switch(ch)                 /* Examine "ch"               */
      {                       /* Body of switch statement   */

         case 'a' :
            printf("You're right!");
            return;

         default  :           /* Any other answer           */
            printf("You're wrong.\n");
            printf("You'll have to try again!");
      }
}
```

Key to Example 4–5:

❶ When using the macro **_tolower**, you must include the definition module ctype.h in the compilation process. The module ctype.h is located in the ULTRIX System Library and defines macros and constructs used for character processing and classification.

In VAX C, the preprocessor directives are processed by an early phase of the compiler, not by a separate program as the name preprocessor implies. Directives, unlike other VAX C lines of source code, begin with a pound sign (#). Do not end preprocessor directives with a semicolon (;). The pound sign must appear in column 1, the leftmost margin of your source file.

The module ctype.h is not the only module that contains macros and definitions used by the functions. There are several ways to include definitions in the program stream. For more information about the VAX C definition modules, see Chapter 9 and the *ULTRIX Documentation Set*.

❷ The compiler replaces the references to the **isupper** and **_tolower** macros with lines of C source code. When the program is run, the value of the variable ch is translated to lowercase but only if it was originally an uppercase letter. To see the macro definitions of **isupper** and **_tolower**, print the file /usr/include/ctype.h from the system library.

Output from Example 4–5 is as follows:

```
% example5 RETURN
Guess which letter I'm thinking of!
A RETURN
You're right!
```

The **switch** statement executes one or more of a series of cases based on the value of the expression in parentheses. If the value of variable ch is a, then the statements following the label case `'a'` are executed. In Example 4–5, the **_tolower** macro in the **if** statement translates any uppercase answers to lowercase letters, so there is no need to test for the uppercase letter A in the **switch** statement. When a case label matches the value of expression ch, the statements following all of the remaining case labels are executed until the compiler encounters a **break** statement (which terminates the immediately enclosing statement), a **return** statement (which terminates the enclosing function), or the bottom of the **switch** statement. The statements following the default label are executed if the value of the expression does not match any of the other case labels. For more information about the **switch** statement, see Chapter 5.

4.5.3 Loops

In the previous examples, the user can only guess once during the execution of the program. To guess another letter, it is necessary to execute the program again. If you want to execute a segment of code repeatedly until a condition is met, you can use a loop. Some loops execute a block of statements, known as the loop body, a specified number of times. Some loops test for a condition first and then execute the body of the loop if the condition is true. Some loops execute the loop body and then test for a condition, which guarantees at least one execution of the body. In VAX C, this last loop is called the **do** statement. Example 4–6 shows that you can use the **do** statement to alter the letter-guessing program.

Example 4–6: Looping Using the do Statement

```
/*  This program plays the same guessing game as the      *
 *  other examples except that the user must guess until  *
 *  the answer is correct.  This is accomplished using a  *
 *  do statement.                                         */

#include <stdio.h>

#include <ctype.h>

main()
{
   char  ch;

   printf("Guess what letter I'm thinking of!\n");
   printf("Keep guessing till you get it!\n");
```

(continued on next page)

Example 4–6 (Cont.): Looping Using the do Statement

```
    do                              /* Do the following ...     */
       {                            /* Beginning of loop body   */
          ch = getchar();
          if (isupper(ch));
          ch = _tolower(ch);
          switch(ch)
             {
                case 'a' :
                   printf("You're right!");
                   return;
                                    /* Ignore RETURN (newline) ch */
❶               case '\n':
                   break;
                default  :
                   printf("You're wrong.\n");
                   printf("You'll have to try again!\n");
             }                      /* End of switch statement  */
       }                            /* End of do loop body      */
                                    /* Condition to be tested   */
❷   while(ch != 'a');
}
```

Key to Example 4–6:

❶ The case label tests to see if the value of the character is a newline character (`\n`). The newline character is entered when you press the RETURN key. If it is the newline character, the character is ignored and a new character is taken from the terminal.

❷ In the **while** expression at the end of the **do** statement, the not-equal-to operator (!=) presents the condition to be tested. The **while** expression translates as follows: "while the variable ch is not equal to `'a'`."

Output from Example 4–6 is as follows:

```
% example6 RETURN
Guess which letter I'm thinking of!
Keep guessing till you get it!
B RETURN
You're wrong.
You'll have to try again!
A RETURN
You're right!
```

The **for** statement allows you to specify the number of times to execute the loop body. In the previous examples, it can be used to limit the number of guesses that the user may attempt. You can use other looping techniques to limit the number of guesses, but you must be responsible for incrementing a counter (the **for** statement increments automatically). Example 4–7 shows the use of the **for** statement.

Example 4–7: Looping Using the for Statement

```
/*  This program plays the same guessing game as the      *
 *  previous examples except that the user is limited to  *
 *  three guesses.  This is accomplished using a for      *
 *  statement.                                            */

#include <stdio.h>

#include <ctype.h>

main()
{
   char     ch;
   int        i;                  /* A counter for loop       */

   printf("Guess what letter I'm thinking of!\n");
   printf("You have three guesses.  Make them count!\n");
                                  /* Do the following 3 times */
❶  for (i = 1; i <= 3; i++ )
      {                           /* Beginning of loop body   */
         ch = getchar();
         if (isupper(ch));
         ch = _tolower(ch);
         switch(ch)
            {
               case 'a' :
                  printf("You're right!");
                  return;
               case '\n':
❷                 --i;
                  break;

               default   :
                  printf("You're wrong.\n");
                  if (i != 3)
                     printf("You'll have to try again!\n");
            }                     /* End of switch statement  */

      }                           /* End of for loop body     */
   printf("Sorry, you ran out of guesses!");
}
```

Key to Example 4–7:

❶ The **for** statement controls how many times the body of the loop is executed. The first expression inside the parentheses following the keyword **for** initializes the loop counter i to value 1. The second expression establishes an upper bound; the value of variable i cannot exceed 3. The third expression establishes the increment or decrement value of the variable that will be executed after every execution of the loop body. The double plus signs (++) are the increment operator; they increase the value of a variable by the integer 1. The loop body is executed, each time the value of variable i increases by 1, until the value of i is greater than 3.

❷ The double minus signs (– –) are the decrement operator. The decrement operator is used in this example to subtract one from the value of variable i so that newline characters are not counted as a guess.

Sample output from Example 4–7 is as follows:

```
% example7 RETURN
Guess which letter I'm thinking of!
You have three guesses.  Make them count!
B RETURN
You're wrong.
You'll have to try again!
```

```
C RETURN
You're wrong.
You'll have to try again!
U RETURN
You're wrong.
Sorry, you ran out of guesses!
```

4.6 Values, Addresses, and Pointers

In VAX C, every variable has two types of values: a memory location and a stored object. In VAX C, an lvalue is the variable's address in memory, and an rvalue is the stored object. Consider the following assignment statement:

```
put_it_here  =  take_this_object;
```

This assignment statement is not very different from statements in other programming languages, but consider the differences between locations in memory and objects stored in memory. This assignment takes take_this_object's rvalue and places it in memory at put_it_here's lvalue.

Consider the following VAX C assignment statement:

```
int  x = 2,  y;

/*  put_it_here  =  take_this_object;   */

         y       =         x;
```

The two distinct variables have different memory locations (lvalues), but, after the assignment statement, they contain objects of the equivalent value 2.

A variable's rvalue can be many things, such as integers, real numbers, character strings, or data structures. One type of rvalue that it can be is the address of another variable. In other words, a variable's rvalue can be another variable's lvalue. In this case, one variable points to another variable.

A declaration of a variable whose rvalue is a pointer to another variable is as follows:

```
int  *pointr;
```

The indirection operator (*) specifies that the variable is a pointer, which in this example points to an object of data type **int**. Pointers are declared as pointing to an object of a particular data type.

You can assign the address of a variable to the pointer as follows:

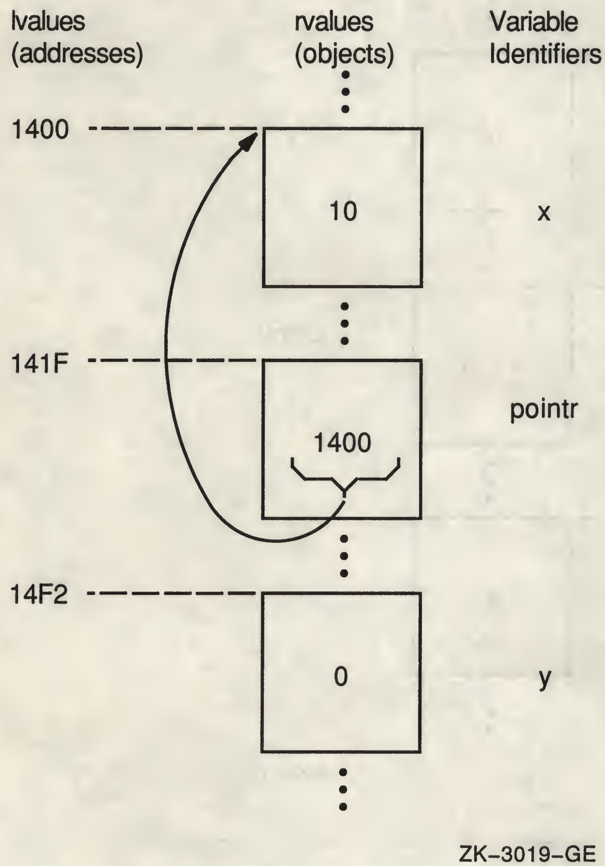
```
static  int *pointr;              /* Declarations              */
static  int x = 10, y = 0;

        pointr = &x;              /* Assignment                */
```

The rvalue of the variable pointr is the lvalue of variable x. In other example assignment statements, the rvalue of the variable on the right side of the equal sign (=) was taken. In this example, the ampersand (&), which is the address of the operator, translates as follows: take the lvalue of this variable instead of its rvalue.

The **static** keyword specifies the **static** storage class. See Chapter 8 for information about **static** and other storage-class specifiers and modifiers.

Figure 4–1 shows the difference between rvalues and lvalues.

Figure 4–1: rvalues, lvalues, and Assigning Pointers



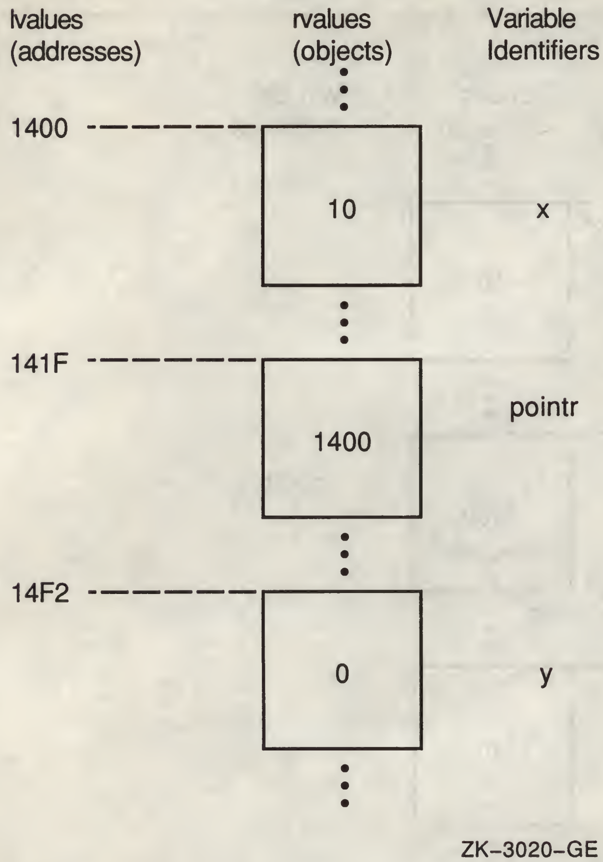
The value of the variable pointr contains the address of variable x. Remember that the location of variables in memory and the order in which the compiler processes them is unpredictable and left to the discretion of the compiler.

After you assign an address to the pointer, you will want to use it. For example, if you want to assign x's rvalue to a variable y, use the pointer in a VAX C statement as follows:

```
y = *pointr;
```

The asterisk (*) is the VAX C indirection operator; the value of the variable being pointed to by pointr is assigned to y. The indirection operator translates as follows: the rvalue of this variable points to some other variable, so go to that location and access the stored object. Figure 4–2 shows the status of the variables after you execute the last code example.

Figure 4–2: The Indirection Operator in Assignments



For more information about pointers, see Chapter 7.

4.7 Aggregates

The variables used in the previous examples were either pointers or single objects that could be manipulated, in their entirety, in an arithmetic expression. These types of variables are called scalar variables. The VAX C data structures (arrays, structures, and unions) are called aggregates. Aggregates are comprised of segments called members. Members are sections of the structure that you can declare to be either a scalar or aggregate data type.

4.7.1 Arrays and Character Strings

An array is a data structure whose members are of the same type. Members of arrays can be any of the scalar or aggregate data types.

VAX C represents character strings internally as arrays of type **char**. A character string may be declared and initialized as a character-string variable using the indirection operator (*), as an array of a specified number of members, or as an array of an unspecified number of members as follows:

```
char *str =       "Hello";
char string[6] = "Hello";
char strng[] =   "Hello";
```

In VAX C, all character strings end with the NUL character (\0). In the previous example, the NUL character is appended to the string Hello to make the string six characters long. When assigning strings to character-string and array variables within the program, you must use the C string-handling functions **strncpy** or **strcpy**. The following example illstrates the use of character strings and arrays.

Example 4–8: Character String Constants and Arrays

```
/*  This program plays the same guessing games as the          *
 *  previous examples except that it uses character            *
 *  string constants and arrays.                               */

#include <stdio.h>

main()
{
   char ch;                          /* Declare a character      */
                                     /* Initialize messages      */
   char  *greeting = "Guess which letter I'm thinking of!";
   char  *message1 = "You're right!";
   char  *message2 = "You're wrong.";
   char  *message3 = "You'll have to try again!";
   char  correct[2];
   correct[0] = 'a';                 /* Store correct letters    */
   correct[1] = 'A';

   printf("%s\n", greeting);         /* %s = char string         */
   ch = getchar();

   if (ch == correct[0] || ch == correct[1])
      printf("%s", message1);
   else
      {
         printf("%s\n", message2);
         printf("%s", message3);
      }
}
```

Output from Example 4–8 is as follows:

```
% example8 RETURN
Guess which letter I'm thinking of!
B RETURN
You're wrong.
You'll have to try again!
```

For more information about arrays and character strings, see Chapter 7.

4.7.2 Structures and Unions

Structures and unions are aggregates whose members can be of different types. Structures and unions are declared using the **struct** and **union** keywords, an optional tag name, and a list of member declarations delimited by braces ({ }). A member of a structure or a union is a declared segment of the data structure. The syntax for declaring a member is the same as for declaring any variable. The structure or union tag is a name that you can use to declare structure or union variables of the same type elsewhere in the program. Members of structures and unions may be referenced as follows:

```
main()
{
   struct  optional_tag                /*  Tag = optional_tag  */
      {
         char   letter_1;
         char   letter_2;
         int    number;
      }  characters = {'a', 'b', 59}; /* Variable = characters */

   characters.letter_1 = characters.letter_2;
}
```

You may reference members by using the structure or union variable name followed by a period (.) or followed by the member name. As in Example 4–8, structures are initialized using a variable name and an assignment operator (=) immediately following the declaration of the members. The values of the members are delimited by braces ({ }) and separated by commas (,). The address of the first member of a structure begins, in memory, at the base of the data structure, which is referred to as offset 0.

Unions are declared in the same way as structures, but all members in a union begin at offset 0. Unlike structures, unions cannot be initialized. The size of the union in memory is as large as its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. Example 4–9 shows the nature of unions.

Example 4–9: Single Storage Allocation of Unions

```
/*  This example shows the storage maintenance of              *
 *  unions with different size members.                        */

#include <stdio.h>

main()
{
   union                            /* Declare the union        */
      {
         char  lastname[8];         /* Array for a last name    */
         char    firstinit;         /* Char. for first initial  */
      }  overlap;
                                    /* Copy and print members   */
   strcpy(overlap.lastname, "Jackson");

   printf("%s\n", overlap.lastname);
   overlap.firstinit = 'M';
   printf("%c\n", overlap.firstinit);
   printf("%s\n", overlap.lastname);
}
```

The output from Example 4–9 is as follows:

```
% example9 RETURN
Jackson
M
Mackson
```

The **strcpy** function copies the second string into the array variable. When assigning values to smaller union members, the compiler does not fill the remaining space in the union with NUL characters (\0); that is, whatever was in memory at the time remains. For more information about structures and unions, see Chapter 7.

Example 4–10 shows a structure definition and its usage.

Example 4–10: Structures

```
/*  This program plays the same guessing game as the      *
 *  previous examples except that it uses a structure.    */

#include <stdio.h>

main()
{
   char ch;
   char  *greeting1 = "Guess which letter I'm thinking of!";
   char  *greeting2 = "You've 3 guesses.  Make them count!";
   char  *message1 = "You're right!";
   char  *message2 = "You're wrong.";
   char  *message3 = "You'll have to try again!";
   char  *message4 = "Sorry, you ran out of guesses!";
   int   i;
                              /* Store information        */
❶  struct  storage            /* Structure tag = storage  */
      {
         char      small_a;   /* One correct letter       */
         char    capital_a;   /* Another correct letter   */
         char   newline_ch;   /* newline character        */
         int   num_guesses;   /* Number of guesses        */
      };

                              /*  Declare "letter"        *
                               *   using tag "storage"    */
❷  struct  storage  letter = {'a', 'A', '\n'};

   letter.num_guesses = 3;
   printf("%s\n", greeting1);
   printf("%s\n", greeting2);

   for (i = 1; i <= letter.num_guesses; i++)
      {
         ch = getchar();
         if (ch == letter.small_a || ch == letter.capital_a)
            {
               printf("%s", message1);
               return;
            }
         else
            if (ch == letter.newline_ch)
               --i;
            else
               {
                   printf("%s\n", message2);
                   if (i != 3)
                      printf("%s\n", message3);
                }

      }                     /* End of for loop body       */
      printf("%s", message4);
}
```

Key to Example 4–10:

❶ The structure declaration with the tag storage has four members. The first three members are of type **char**. The last member is of type **int**.

❷ The struct variable letter is declared using the tag storage. At the same time, individual members of the structure are initialized. The equal sign initializes the members of the structure variable with constants. The constants are

separated by a comma and are delimited by braces. The number of initializing constants cannot exceed the number of members. However, as in this example, you may omit constants; the compiler pads the uninitialized member (in the example, member num_guesses) with zeros. However, you cannot initialize a member in the middle of any aggregate without initializing the previous members.

Output from Example 4–10 is as follows:

```
%  example10 RETURN
Guess which letter I'm thinking of!
You've 3 guesses.  Make them count!
B RETURN
You're wrong.
You'll have to try again!
C RETURN
You're wrong.
You'll have to try again!
U RETURN
You're wrong.
Sorry, you ran out of guesses!
```

See Chapter 2 and Chapter 3 for information about developing programs on ULTRIX systems.

4.8 Function Definitions

You must declare or define functions that you want to call or use in a VAX C program. You may or may not need to declare user-defined functions before you call them. This depends on what type of value the function returns, and the position of the function definition within the program. This section explains the rules for defining functions, but you may want to read the discussion of declarations and definitions in Chapter 7 before proceeding.

In a function definition, you specify the VAX C statements that execute whenever you call the function. You also specify the parameters (if any) of the function. The parameters of a function provide a means to pass data to the function. See Section 4.11 for a detailed discussion of parameters and arguments.

Example 4–11 shows an example of two function definitions.

Example 4–11: Case Conversion Program

```
/*  This program converts its input to lowercase.  The    *
 *  first function passes control to the second function  *
 *  to convert a letter.  Comments are located to the     *
 *  right of the code.                                    */

#include <stdio.h>             /* To use I/O definitions    */
main()
❶{
   FILE *infile, *outfile;    /* Declare files             */
   int        i, c, c_out;
                              /* Open "infile" for input   */
   infile =  fopen("ex113.in", "r");

                              /* Open "outfile" for output */
   outfile = fopen("ex113.out", "w");
```

(continued on next page)

Example 4–11 (Cont.): Case Conversion Program

```
                                   /* While not end of file...  */
                                   /* Get a char from the file  */
   while ((c = getc(infile)) != EOF)
       {
       c_out = lower(c);           /* Send char to "lower"      */
                                   /* Output the char to file   */
       putc(c_out, outfile);
       }
   return;                         /* Optional return statement */
}

/* --------------------------------------------------    *
 *  Beginning of the next function definition:           *
 * --------------------------------------------------    */
                                    /* Function and parameter   *
                                     *   name                   */
❷ lower(c_up)
❸ int c_up;                         /* Declare parameter type   */
{                                   /* Beginning function body  */
                                    /* If capital, convert      */
   if (c_up >= 'A' && c_up <= 'Z')
      return c_up - 'A' + 'a';
   else                             /* Else, return as is       */
      return c_up;
}                                   /* End of function body     */
                                    /* End function definition  */
```

Key to Example 4–11:

❶ Program execution begins with function main. The left brace ({) signifies the beginning of the function body; the right brace (}) signifies the end of the body. The function body is any set of valid VAX C statements or declarations. The body usually includes one or more **return** statements, as shown here. A **return** statement specifies an expression whose value is returned to the calling function. If you omit the expression, the returned value is undefined in the calling function. If you omit the **return** statement, the function terminates when the right brace is encountered and its return value is undefined. See Chapter 5 for more information about the **return** statement.

❷ The **lower** identifier begins a new function definition; lower has the single parameter c_up. Function main has no parameters, but the parentheses must be present.

❸ The next statement, int c_up, declares the parameter's data type, in this case, **int** (integer). The declaration is omitted if the function has no parameters; furthermore, declarations here in the program should specify only the names of parameters, not the names of other variables used in the function body. See Chapter 7 for more information about data types.

For more information about the VAX C operators used in Example 4–11, see Chapter 6.

4.8.1 The main Function and Function Identifiers

The execution of a program begins at the function whose identifier is main, or, if there is no function with this identifier, at the first function seen by the linker. In Example 4–11, the main function physically precedes the **lower** function, but the two function definitions can appear in reverse order. The word main is not a language keyword, so you may use it for other purposes in the program.

Function names have compile-time scope rules that are slightly different from those that apply to other identifiers. Any valid function identifier followed by a left parenthesis is declared implicitly as the name of a function whose storage class is external and whose return value is of the data type **int**. For more information about scope and storage classes, see Chapter 8.

Between the definition of a function's identifier and the declaration of its parameters, you can write the following option:

```
main_program
```

The main_program option identifies the function as the main function in the program. It is not a keyword, and it can be expressed in either upper- or lowercase. Use the main_program option if the program does not contain function main, and if you do not want the program's execution to begin at the first function linked. For example, the following definition establishes function lower as the main function; execution begins there, regardless of the order in which the function is linked:

```
char lower(c_up)
MAIN_PROGRAM
int c_up;
{
    .
    .
    .
}
```

NOTE

The main_program option is VAX C specific and is not portable.

4.8.2 Parameter List Declarations

Example 4–11 shows only one of two possible methods of listing function parameters, as follows:

```
lower( c_up )
int  c_up;
{
    .
    .
    .
```

To make your code concise, you may want to list the data types of the function parameters within the parameter list. If you use this method, your function definition also serves as a function prototype. See Section 4.10 for more information about the effect of function prototypes.

The second way to declare parameter data types is as follows:

```
lower( int c_up )
{
    .
    .
    .
```

For instance, if you need to declare parameters of different data types, your function definition may appear as follows:

```
function_name( int lower, int upper, int temp, char x, float y )
{
    .
    .
    .
```

If you are using the function prototype format in a function definition, you must supply both an identifier and a data type specification for each parameter. If you do not, the action generates an error message.

In a function definition, you have the following two options when specifying an empty parameter list:

- You can specify empty parentheses.
- You can use the **void** keyword to specify an empty parameter list.

An example using the **void** keyword is as follows:

```
char function_name( void )
{ return 'a'; }
```

4.8.3 Function Return Data Types

By default, all VAX C functions return objects of data type **int**. In Example 4–11, function lower returns an integer to the main function using the **return** statement.

If you define a function that returns anything other than an integer, you need to specify the function return data type in the function definitions and declarations. The following example shows the definition of a function returning a character:

```
char letter( int param1, char param2, int *param3  )
{
    .
    .
    .
    return param2;
```

If a function does not return a value, or if you do not call the function within an expression that requires a value, you can define the function to return a value of type **void**. Using the **void** keyword generates an error under the following conditions:

- If the function returns a value.
- If you call the **void** function in an expression that requires a return value.
- If you use the **void** keyword with the cast operator for anything other than a function.

The following example shows the use of the **void** keyword to specify a function without a return value and to specify a null parameter list:

```
void  message( void )
{
   printf("Stop making sense!");
   return;
}
```

4.8.4 Variable-Length Parameter Lists

If you decide to define a function with a variable-length parameter list, you can use ellipses (. . .) to designate the variable-length portion of the parameter list, as follows:

```
function_name( int lower, int upper, int temp, char x, float y, ... )
{
   .
   .
   .
```

Within the function body, use the varargs functions and macros to access the argument list passed to the function. The varargs functions and macros provide a portable means of accessing variable-length argument lists. For more information about variable-length argument lists, see the description of the standard include file varargs.h in the *ULTRIX Documentation Set*.

When using ellipses for variable-length argument lists, you must have at least one argument preceding the ellipses. The following definition is allowed:

```
function_name( double lower, ... )
{
   .
   .
   .
```

The following definition is not allowed:

```
function_name( ... )
{
   .
   .
   .
```

If you are not using function prototypes, you can use the varargs header and declaration within the parameter list and before the function body, instead of using the ellipsis notation. The following example shows such a construct:

```
function_name( va_alist )
va_dcl
{
   .
   .
   .
```

NOTE

If you use function prototypes, use ellipses (. . .) within parameter lists so that the compiler does not compare varargs declarations (va_alist and va_dcl) with prototype data declarations. See Section 4.10 for more information about function prototypes.

4.9 Function Declarations

As in Example 4–11, you can call a function without declaring it, if the function's return value is an integer. If the return value is anything else, you may have to declare the function. Example 4–12 shows a case where you need to declare a function.

Example 4–12: Declaring Functions

```
#include <stdio.h>

main()
{
❶   char lower();           /* The function declaration */
        .
        .
        .
    while ((c = getc(infile)) != EOF)
        {
                            /*  The function call       */
          c_out = lower(c);
          putc(c_out, outfile);
        }
}

char lower(c_up)            /* The function definition  */
int c_up;
{
    .
    .
    .
}
```

Key to Example 4–12:

❶ Since the location of the function definition is located after the main function in the source code, and since the **lower** has a return type of **char**, you have to declare the function before calling it.

If the function definition of **lower** is located before the main function in the source code, despite **lower**'s return data type, you do not need to declare the **lower** before you call the function.

In a function declaration, you can use the **void** keyword to specify an empty argument list, as follows:

```
main()
{
   char function_name( void );
        .
        .
        .
}
char function_name( void )
{ }
```

If the function's return data type is **void**, you must use the keyword in the declaration, as follows:

```
main()
{
   void function_name( void );
         .
         .
         .
}
void function_name( void )
{ }
```

If you specify argument data types or **void** in a function declaration, as shown in the following example, VAX C treats the function declaration as a function prototype for the scope of the declaration:

```
main()
{
   char function_name( int x, char y );
         .
         .
         .
}
```

Since the declaration is within the scope of function main, VAX C uses the function declaration as a function prototype only within function main. See Section 4.10 for more information about function prototypes.

4.10 Function Prototypes

A function prototype is a function declaration that specifies the data types of its arguments in the identifier list. VAX C uses the prototype to ensure that all function definitions, declarations, and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

Function prototypes provide compile-time argument checking similar to that found in the lint utility. See Section 4.16 for more information.

When using function prototypes, you can first define the following function:

```
char  function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

You can also define the function as follows:

```
char  function_name( lower, upper, func, y )
int   lower;
int   *upper;
char  (*func)();
double y;
{ }
```

This function's identifier list includes an integer, a pointer to an integer, a pointer to a function returning a character, and a double floating point value. The type specifications are identical to the ones used in a parameter list located before the function body. For more information about interpreting complex declarations, see Chapter 7.

In each compilation unit in your program, you should determine where to place the corresponding function prototype. The position of the prototype determines the prototype's scope; the scope of the function prototype is the same as the scope of any function declaration. VAX C checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. If you misplace the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype, the results are undefined.

Corresponding function prototype declarations are identical to the header of a function definition that specifies data types in the identifier list. Since prototypes are actually function declarations, end the prototype code with a semicolon (;). The following example shows a prototype that corresponds with either of the previous function definitions:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
```

When declaring function prototypes, you do not need to use the same parameter identifiers as in the function definition. If you choose, you do not need to specify any identifiers in the prototype declaration. The scope of the identifiers within function prototypes exists only within the identifier list; you are free to use those identifiers outside of the prototype.

For example, you can use any of the following prototype declarations for the function definition presented:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
char  function_name( int a, int *b, char (*c)(), double d );
char  function_name( int, int *, char (*)(), double );
```

You can specify variable-length argument lists in function prototypes by using ellipses. You must have at least one argument in the list preceding ellipses. The following example shows the specification of a variable-length argument list:

```
char  function_name( int lower, ... );
```

You cannot omit data type specifications in a function prototype, nor can you have a variable-length argument list that is not preceded by at least one argument. The following prototypes are not allowed and their use generates appropriate error messages:

```
char  function_name( lower, *upper, char (*func)(), float y );
char  function_name( , , char (*func)(), float y );
char  function_name( ... );
```

Using the function prototype ensures that all corresponding function definitions, declarations, and calls within the scope of the prototype conform to the number and type of parameters specified in the prototype. A function prototype is considered in scope only if a function prototype declaration is specified within a block enclosing the function call or at the outermost level of the source file. If a prototype is in scope, the automatic widening of **float** arguments to **double** is not performed. However, the automatic widening of **char** and **short int** arguments to **int** is performed. If the number of arguments in a function definition, declaration, or call does not match the prototype, the statement generates the appropriate message.

If the data type of an argument in a function call does not match the prototype, VAX C tries to perform conversions. If the mismatched argument is assignment compatible with the prototype parameter, VAX C converts the argument to the data type specified in the prototype, according to the parameter and argument conversion rules (see Section 4.11).

If the mismatched argument is not assignment compatible with the prototype parameter, the action generates the appropriate error message and the results are undefined.

The syntax of the function prototype is designed so that you can extract the first line of each of your function definitions, add a semicolon (;) to the end of each line, place the prototypes in a .h definitions file, and include that file at the top of each compilation unit in your program. In this way, you declare the function prototypes to be external, so that the scope of the prototype extends throughout the entire compilation unit. You place the include preprocessor directives at the top of any applicable compilation units.

See Chapter 9 for more information about preprocessor directives. See Chapter 8 for more information about compilation units and scope.

4.11 Using Parameters and Arguments

VAX C functions can exchange information by using parameters and arguments. (In this guide, the term parameter means the variable within parentheses named in a function definition; the term argument means an expression that is part of a function call.) In Example 4–11, function lower has the single parameter c_up. When this function is called from the main function, argument c is evaluated and passed to function lower.

The following rules apply to the parameters and arguments of VAX C functions:

- The number of arguments in a function call must be the same as the number of parameters in the function definition. This number may be zero.
- In VAX C, the maximum number of arguments (and corresponding parameters) is 253 for a single function. The maximum length of an argument list is 255 longwords.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the compiler may evaluate the arguments in any order. Do not expect function calls or other complicated expressions in the argument list to be evaluated in any particular order.
- In VAX C, all arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value, not its address. The rule applies to all scalar variables, structures, and unions passed as arguments. A function cannot modify the values of its arguments. However, since arguments can be addresses or pointers, a function can use addresses to modify the values of variables defined in the calling function.
- The types of evaluated arguments must match the types of their corresponding parameters. When a function is called, unless a function prototype is in scope, VAX C does not compare the types of the arguments with those of the corresponding parameters so it does not generally convert the arguments to the types of the parameters. Instead, all of the expressions in the argument list are converted according to the following conventions:
  - Any arguments of type **float** are converted to **double**.
  - Any arguments of types **char** or **short** are converted to **int**.
  - Any arguments of types **unsigned char** or **unsigned short** are converted to **unsigned int**.
  - Any function name appearing as an argument is converted to the address of the named function. The corresponding parameter must be declared as a pointer to a function, which evaluates to a value of the same data type as the function.
  - Any array name appearing as an argument is converted to the address of the first element of the array. The corresponding parameter can be declared either as an array of the given type or as a pointer to the given type. Since character-string constants are declared implicitly as arrays of characters, this rule also applies to the use of string constants as arguments.

  No other conversions are performed on arguments. If you know that a particular argument must be converted to match the type of the corresponding

parameter, use the cast operator. For more information about the cast operator, see Chapter 6.

- If you declare variables in the parameter declaration section that do not exist in the parameter list, these variables are treated as if they are declared in the function body. However, this is not good programming practice and, if used, your programs may not be portable.
- If you do not declare parameters, they are implicitly declared to be of data-type **int**.

4.11.1 Function and Array Identifiers as Arguments

You can use a function identifier without parentheses and arguments. In this case, the function identifier evaluates to the address of the function. This method of referencing is useful when passing a function identifier in an argument list. You can pass the address of one function to another as one of the arguments.

If you want to pass the address of a function in an argument list, the function must either be declared or defined, even if the return value of the function is an integer. Example 4–13 shows when you must declare user-defined functions and how to pass functions as arguments.

Example 4–13: Declaring Functions Passed as Arguments

```
                                /* Defined before it is      *
                                 *  used                     */
❶ x()  { return 25; }
main()
{
❷   int y();                    /* Function declaration      */
       .
       .
       .
❸   funct(x, y);                /* Passed as addresses       */
       .
       .
       .
}

y() { return 30; }              /* Function definition       */

funct(f1, f2)                   /* Function definition       */
                                /* Declare arguments as      *
                                 *  pointers to functions    *
                                 *  returning an integer     */
❹ int (*f1)(), (*f2)();
{
   (*f1)();                     /* A call to a function      */
       .
       .
       .
}
```

Key to Example 4–13:

❶ You can pass function x in an argument list since its definition is located before the main function.

❷ You must declare function y before you pass the function in an argument list since its function definition is located after the main function.

❸ When you pass functions as arguments, do not include the parentheses.

❹ When declaring the function identifiers as parameters, declare the function as the result of the indirection operator (*) applied to the address of the function. For more information about parentheses in expressions and the indirection operator, see Chapter 6.

VAX C treats array parameters in the same way. If you pass an array identifier in an argument list, VAX C translates the identifier as a pointer to the data type of the array elements. To access the first element of the array, you need to dereference the pointer. For more information about pointers, addresses, and dereferencing, see Chapter 7.

4.11.2 Passing Arguments to the main Function

The main function in a VAX C program can accept arguments from the command line from which it was invoked. The syntax for a main function is as follows:

```
int main(argc, argv, envp)
int argc;
char *argv[ ],*envp[ ];
```

In this syntax, parameter argc is the count of arguments present in the command line that invoked the program, and parameter argv is a character-string array of the arguments. Parameter envp is the environment array. It contains process information, such as the user name and controlling terminal. It has no bearing on passing command-line arguments. Its primary use in VAX C programs is during **exec** and **getenv** function calls.

In the main function definition, the parameters are optional. However, you can access only the parameters that you define. You can define function main in any of the following ways:

```
main()
main(argc)
main(argc, argv)
main(argc, argv, envp)
```

Example 4–14 shows a program called echo.c, which displays the command-line arguments that were used to invoke it.

Example 4–14: Echo Program Using Command-Line Arguments

```
/*  This program echoes the command-line arguments.          */

#include <stdio.h>

main(argc, argv)
int  argc;
char *argv[];
{
   int i;
                                   /* argv[0] is program name  */
   printf("program: %s\n",argv[0]);

   for (i = 1;  i < argc;  i++)
      printf("argument %d: %s\n", i, argv[i]);
}
```

You can compile and link Example 4–14 using the following command:

```
% vcc  -o ECHO echo.c RETURN
```

Sample output from Example 4–14 is as follows:

```
% ECHO  Long  "Day's"  "Journey into Night" RETURN
program: /usr/oneill/plays/ECHO
argument 1: Long
argument 2: Day's
argument 3: Journey into Night
```

4.12 Identifiers

Identifiers can consist of letters, digits, dollar signs ($), and the underscore character (_). Do not create identifiers with a length of more than 255 characters. If you do, the compiler truncates the name and generates a warning message.

The first character must not be a digit, and to avoid conflict with names used by VAX C, should not be an underscore character. VAX C uses a preceding underscore to identify most implementation-specific macros and keywords, and uses two preceding underscores to identify implementation-specific constants.

Upper- and lowercase letters specify different variable identifiers; that is, the compiler interprets abc and ABC as different variable names.

Use the following conventions if practical:

- Avoid using underscores as the first character of your identifiers.
- Type identifiers in uppercase if they are constants that are given values by the **#define** directive.
- Type all other identifiers and keywords in lowercase.

4.13 Keywords

Keywords are predefined identifiers. They cannot be redeclared. They identify data types, storage classes, and certain statements in VAX C. Many conventional words in VAX C programs are not keywords and can be redeclared. The notable examples are the names of functions, including main and the functions found in system libraries.

Keywords must be expressed in lowercase letters.

Table 4–1 lists the VAX C keywords.

Table 4–1: VAX C Keywords

Keyword	Meaning
Type Specifiers	
int	Integer (on a VAX, 32 bits)
long	32-bit integer
unsigned	Unsigned integer
short	16-bit integer
char	8-bit integer
float	Single-precision, floating-point number
double	Double-precision, floating-point number
struct	Structure (aggregate of other types)
union	Union (aggregate of other types)
typedef	Tagged set of type specifiers
enum	Enumerated scaler type
void	Function return type
variant_struct[1]	Variant structure
variant_union[1]	Variant union
Storage-Class Specifiers	
auto	Allocated at every block activation
static	Allocated at compile time
register	Allocated at every block activation
extern	Allocated by an external data definition (at compile time)
globaldef[1]	Definition of a global variable
globalref[1]	Reference to a global variable
globalvalue[1]	Definition or declaration of a global value
readonly[1]	Allocated in read-only program section
noshare[1]	Assigned NSHR program section attribute
_align[1]	Aligned on a specified boundary
Type Modifiers	
const	Object cannot be modified
volatile	Object cannot be assigned to a register
Statements	
goto	Transfers control unconditionally
return	Terminates a function and optionally returns a value to the caller
continue	Causes next iteration of the containing loop

[1]VAX C specific and nonportable.

(continued on next page)

Table 4–1 (Cont.): VAX C Keywords

Keyword	Meaning
Statements	
break	Terminates its corresponding **switch** or loop
if	Executes the following statement conditionally
else	Provides an alternative for the **if** statement
for	Iterates the next statement (zero or more times) under control of three expressions
do	Iterates the next statement (one or more times) until a given condition is false
while	Iterates the next statement (zero or more times) while a given expression is true
switch	Executes one or more of the specified cases (multiway branch)
case	Begins one case for **switch**
default	Provides the default case for **switch**
Operator	
sizeof	Computes the size of an operand in bytes

The following identifiers are not true keywords, but the VAX C compiler defines substitutions so do not redefine them:

```
vax        VAX
vaxc       VAXC
vax11c     VAX11C
unix
ultrix
bsd4_2
CC$gfloat
```

See Chapter 9 for more information about these identifiers.

4.14 Blocks

A block is a compound statement surrounded by braces ({ }). You can use a block wherever the grammar of VAX C requires a single statement. The common cases are the bodies of functions and **if**, **for**, **do**, **switch**, and **while** statements. This definition of a block may conflict with its definition in other languages. In VAX C, the terms block and compound statement are identical.

A block may also contain declarations. If it does, any declarations of **auto**, **register**, or **static** variables declare names that are local to the block. Example 4–15 presents nested blocks and the differences in the scope of declared variables.

Example 4–15: Scope of Variable Declarations in Nested Blocks

```
/*  This program shows how variables with the same        *
 *  identifier can be of different data types if they     *
 *  are located in different blocks.                      */

main()
{                                 /* Outer block of "main"   */
❶   int i;
    i = 1;
       .
       .
       .
    if (i == 1)
       {                          /* An inner block          */
❷         float i;
             .
             .
             .
          i = 3e10;
       }
}
```

Key to Example 4–15:

❶ In all blocks of the program, except the block in the **if** statement, variable i is an integer. The default storage class for this variable is **auto**.

❷ Within the block in the **if** statement, variable i is a single-precision floating-point value. Since it is also of the storage class **auto**, a new floating-point version of variable i is allocated each time the inner block is activated.

If initialization is specified for any **auto** or **register** variables in a block, it is performed each time control reaches the block normally; that is, such initializations are not performed if a **goto** statement transfers control into the middle of the block or if the block is the body of a **switch** statement. For more information about data types, see Chapter 7. For more information about scope and storage classes, see Chapter 8.

4.15 Comments

Comments, delimited by the character pairs (/*) and (*/), can be placed anywhere that white space can appear. The text of a comment can contain any characters except the close-comment delimiter (*/). You cannot nest comments.

4.16 Source Code Checking Functionality

The lint utility provides a way to check source code for improper definitions and declarations, for parameter and argument mismatching, and for inefficient coding practices. VAX C provides the following features that, when combined, offer much of the functionality found in the lint utility at compile time:

Feature	Description
–V standard=portable	When you use the **vcc** command to compile your source code, add this option to the command. The compiler flags constructs that may not be supported by other implementations of the C language.
Function Prototypes	The use of function prototypes allows VAX C to check the number and the data types of all arguments passed to functions. See Section 4.10 for complete information.